TAKING AN OBJECT-ORIENTED VIEW OF ACCELERATORS*

Hiroshi Nishimura, Lawrence Berkeley Laboratory, University of California, CA 94720 USA

It was almost a decade ago that accelerator experts were introduced to the concept of object-oriented programming. This new methodology was expected to play a key role in solving various software problems. Today, there seems be nothing that prevents us from taking advantage of this new technology. In fact, we are often forced to catch up with the computer industry's new *standards*. This is especially true for graphics programming.

While evaluating the benefits of using this new technology, we must also evaluate whether it is as effective as it was claimed. In this paper, we discuss these issues based on experiences at the Advanced Light Source[1].

I. INTRODUCTION

An accelerator project has a life cycle covering the design, construction and operation phases. Each stage needs various kinds of software supports. The role of software construction becomes successively more important from stage to stage.

The design phase is focused on specifications that determine when and how the actual construction starts and operates. Software construction has a procedure compatible with the structured approach, which is supported by structured programming and sometimes enforced by structured analysis and design. Sometimes the use of the structured approach is not directly related to software construction. When a project has specification-based procedures it indicates that the project management is structured.

The need for a structured approach is evident in hardware construction projects, such as buildings, magnets, and power supplies, because it is very difficult to modify them after they have been created. This need is not always as clear in software development, since software can allow for more flexibility. Using a structured approach may sacrifice the software's flexibility. In the early stages of a project the software specifications are in a conceptual state and tend to be incomplete especially for high level application programs. In addition, the rapid progress of computer technology may make a specification obsolete much earlier than expected.

Lack of flexibility also makes software management difficult in the normal operation phase. The duration of the accelerator operation is much longer than the use of specific software, so there is a continuous demand for the software modifications. This is particularly true for synchrotron light sources which have a wide variety of uses and operation modes. The software system must be designed to accommodate changes during the operation. This is a relatively new requirement.

* This work was supported by the Director, Office of Energy Research, Office of Basic Energy Sciences, Material Sciences Division of the U. S. Department of Energy under Contract No. DE-AC03-76SF00098.

Although the structured approach has improved software quality and productivity, a better method must be cultivated.

II. OBJECT-ORIENTED APPROACH

The situation described above is not limited to accelerators. Object-oriented programming (OOP) is believed to be panacea by many developers[2]. There are two approaches in OOP: a pure approach like Smalltalk and a hybrid approach like C++. Today, the hybrid approach, the most common of which is C++, has prevailed not only for accelerators, but in many other fields as well.

The merits of adopting OOP have been discussed frequently in the literature and are best described in "Object-Oriented Software Construction", by B. Meyer[3]. Since there are many aspects to OOP, we will discuss only a few.

A. Modularity

Increased modularity is the most immediate and natural result of OOP. It is analogous to the use of ICs in hardware circuits[4]. This feature encourages the separation of module developers and module users. Here a module means a class library. A developer can create modules without knowing the final requirements of the user. The user can utilize modules without knowing much about their implementation detail.

B. Flexibility

Software development is often limited by its complexity, but this can be significantly improved by more modularity. In addition, users have the freedom to customize the modules for their own purposes through an inheritance mechanism. This flexibility reduces the role of rigid software specifications and can provide more expandability and adaptability.

III. AREA OF OBJECT-ORIENTATION

OOP has become popular in the graphical user interface (GUI) field of programming. This has happened much earlier than in other fields because GUI is complex enough to encourage developers to adopt OOP. Although GUI plays an important role in accelerator-related programming, we have not discussed it here because it is not specific to our field. We have focused on the following items:

A) Accelerator modeling and simulation
B) Device control
C) Machine studies
D) Machine operations

A. Accelerator Modeling and Simulation

The most direct method of treating accelerator components, such as magnets, drift space and beam position monitors (BPMs), is as objects. This is because there are always real physical objects associated with them. By combining such elements or objects we can create a structured object to represent a beam-line as a series of the component objects. The circular ring can be a special type of a beam-line object. Accelerator modeling and simulation are very basic subjects for OOP.

Whether to use a beam object or not, becomes a design issue. There is not a lot of merit in treating a single particle as an object, but a bunch with multiple particles can be effectively simplified into an object.

Operator overloading does not always require the use of OOP, but can be supported efficiently by some of the OOP languages, such as C++. This mechanism has been popular with complex, vector and matrix calculations and is easily extendible to cover automatic differentiation or differential algebra (DA)[5]. In the same way, the accelerator-lattice definition can be simplified. Once we have found that the relationship between a lattice and its magnets is the same as that of a string and its characters. A string class can be extended to support the lattice definition.

B. Device Control

Device control involves the construction of a virtual device-class library on top of a device-access library that is not object oriented. A physical device is usually associated with multiple access channels. It is the application developer's task to deal with its details. A virtual device layer serves to hide the intrinsic complexity and provide flexibility through the inheritance mechanism.

C. Machine Study

Machine study is a special mode of machine operation performed by accelerator experts. Accelerator experts have a continuous need for software development and often become part of the study. In addition, accelerator experts who are not software developers will develop programs using whatever method they can use. Ideally they can share software resources as class libraries.

D. Machine Operation

Quite contrary to the case of machine studies, machine operation is carried out by operators. Programs for operation are provided by the control software experts. Therefore, it should be possible to adopt OOP and create class libraries. These class libraries will eventually provide a standard for the control programs.

The key issue is the execution management of the programs for operators. A control system is not a collection of individual programs, and there is a need for a mechanism to manage the overall operational context for the machine, for example, with regard to locking and unlocking devices, according to the operation mode. But this is not sufficient to accommodate very complex operation mode. One possible approach is to recognize operations as objects. Then the problem becomes a run-time management of these operation objects. There is a possibility that we can reuse some kind of management scheme that has been created in other fields.

IV. OBJECT-ORIENTATION AT ALS

The ALS is small enough to adopt OOP, through a small number of software developers, at an early phase of its construction. OOP has been constantly used for GUI programming, modeling and simulation, and device controls since the ALS commissioning phase[6].

A. Accelerator Modeling and Simulation

As one of the first projects of the third generation light sources, the importance of accelerator modeling and simulation studies was strongly emphasized in the lattice design phase[7]. We have developed modeling and simulation programs, a 4x5 matrix code Tracy[8] and a full 6x6 code Gemini[9], using structured programming in Pascal. As these programs use the Pascal-S compiler[10] as the framework, users could freely program their procedures in Pascal to carry out simulation studies. The expandability and compatibility was not enough to develop a high level application program with the ALS control system[11].

During the commissioning phase, the kernel of Tracy was extracted and rewritten as a portable C library. Then OOP was applied by using Eiffel just on top of the C library[12]. Later, the library was completely redesigned in C++ to be compatible with the control system. This C++ class library is now called Goemon.[13]

Accelerator components such as magnets and BPMs are clearly separated from structures such as beam lines and rings. The component class has a class structure shown in Figure 1.

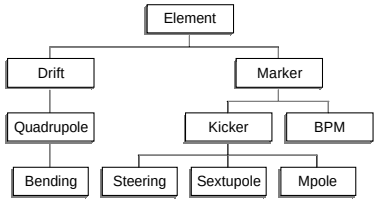

Figure 1. Goemon Component Class Library

Notice that a wiggler or an undulator is an exception, because it is represented either by a hard-edge model or a vertical quadrupole, and it is a composite component that is a beam line by itself.

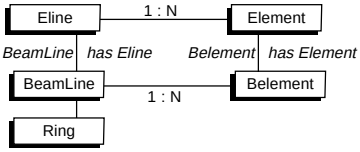

Figure 2. Goemon Structure Class Library

Figure 2 shows the Goemon structure class. Eline is a sequence of the Element objects used to define a lattice

structure by using an operator without using a parser as shown in Table 1. Element is the root of the component class.

```
DRIFT(L1,3.378695);
DRIFT(L2,0.434500);
.....................
BEND( B,0.43257, 5.00, 3.00, 0.00, -0.810);
.....................
Eline CELL=SYM+L1+2*QF+L2+2*QD+L3+B+L4+2*QFA
          +L5+B +L5+2*QFA+L4+B+L3+2*QD+L2+2*QF+L1;
```

Table 1. Lattice Definition using Operator Overloading

Belement is one unit of a beam line that refers to Element and has storage for the optics functions and particle orbits at that location. BeamLine represents a beam-line and uses Eline to read the lattice structure and supports particle tracing and optics calculations.

Ring is a special BeamLine for circular accelerators. ALS structures are supported by derived classes with customized creators. The library also supports a full 6x6 tracking routine to simulate the effect of closed-orbit path-lengthening.

B. Device Control

Single-Device Control The ALS control system uses PCs that run Windows as operator consoles. The application programs were developed on them. Each physical device has one or more DMM channels. A DMM channel is a flat entry to the device-control channel-arrays. Each DMM channel has several subchannels to get and/or set process values, monitor values and Boolean values.

A DMM channel is wrapped with an object called DMMobj[14] that is a root class of single virtual devices. It was originally developed in Pascal with object support and rewritten in C++ on Windows 3.1. Then it was ported to Windows NT. Therefore, one device-class library could be kept for several implementations. Table 2 is an example for the declaration of a DMMobj in C++:

```
class DMMobj
{
  public:
    UBYTE2 errCode;
    UBYTE4 index;
    DMMobj();
    DMMobj(char *aName);
   ~DMMobj();
    void findName(char *aName);
    virtual float getAM(void);// get monitor value
    virtual float getSP(void);// get process value
    virtual void  setSP(float aSP);// set process value
    virtual int   getBM(void);// boolean monitor
    virtual int   getBC(void);// get boolean
    virtual void  setBC(int OnOff);// set boolean
    //   block transfer
    UBYTE2 getOffset(UBYTE2 Control);
    void getBytes(UBYTE2 Offset,UBYTE1*s,int n);
    void setBytes(UBYTE2 Offset,UBYTE1*s,int n);
};
```

Table 2. DMMobj Header in C++

The class structure under DMMobj is shown in Figure 3. Each physical device is supported by one of the classes in this family. Magnets, BPMs, and vacuum gauges are supported by corresponding classes that do not depend on the accelerator section where a single-device would be located. Undulators, DCCT beam-current monitor, and RF systems are supported by specialized classes[15] at the device level.

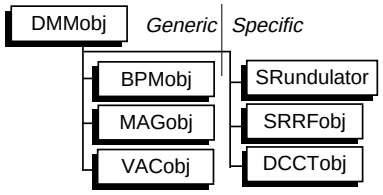

Figure 3. Single Device Class

Multiple-Device Control Multiple-device control is needed to control a group of devices collectively. Figure 4 shows the orthogonal relationship between a single-device class and a multiple-device class. For example, BPMobj supports an individual BPM in any accelerator section, and the multiple device class SRBPMS supports all the BPMs in the storage ring. A multiple device class is fully customized for the section it locates. In addition it supports quasi-synchronized, grouped device access and file access.

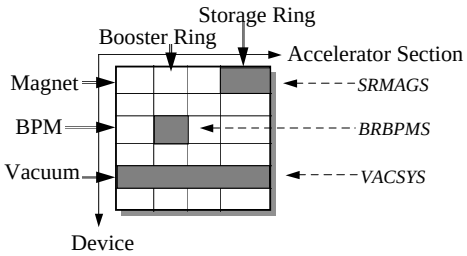

Figure 4. Single and Multiple Devices

Table 3 shows the header of SRBPMS in C++:

```
class SRBPMS
{
   BPMobj *BPM[97];
 public:
   // for data exchange
  SRBPMrec BPMrec;
  SRBPMS();
 ~SRBPMS();
  void  getData();
  // access the n-th BPM
  float getX(int n);
  float getY(int n);
  float getXave(int n);
  float getYave(int n);
  float getXref(int n);
  float getYref(int n);
  float getXraw(int n);
  float getYraw(int n);
  void  calcStat();
  void  setRef();
  void  setOffset();
  void  clearRef();
  void  write(FILE *f);
  void  write(char * fname);
  void  read(FILE *f);
  void  read(char * fname);
  };
```

Table 3. SRBPMS Header in C++

The function getData() does a quasi-synchronized reading of all the storage-ring BPMs. As each BPM reading takes 2.0 msec to read X and Y, getData() takes about 200 msec for 96 BPMs. After using getData() functions getX(n) and getXave(n), etc., are available for access to individual data. This class also supports averaging and offset manipulation.

These device classes have been in daily use to support all the magnets, BPMs, and vacuum readings in the 1.5 GeV injection beam line and the storage ring. These magnets are still operated by using Pascal programs written during the commissioning phase. Other programs that involve BPMs were written in C++ and have been moved to Windows NT.

These multiple device classes must interface with other classes, such as the simulation class. Instead of pointing to

the other class, we use a data class to communicate indirectly. In the case of SRBPMS, this class called SRBPMrec. It is also shared by a simulation class and a graphics class.

EPICS Channel-Access Currently, all the machine devices are controlled by using the original DMM-based control system, except for the communication part of the undulator feedforward-programs[16]. This is because EPICS[17] channel access is used for the photon beam line control. Channel access has been ported to PCs running Windows NT and is supported as an object in C++[18].

C. Machine Studies

As machine studies are carried out on the PCs running Windows, there is a wide variety of software tools available for physicists and engineers who actually program. The tools actively used are LabView, MatLab, and compilers.

LabView is mainly used to control GPIB devices for instrumentation purposes[19], MatLab for the area where the algorithm itself has to be developed and compilers where high performance is required. The compiled applications are mostly written in C++ on Windows 3.1 or Windows NT 3.5.

D. Machine Operation

Although there are 3 insertion devices in the ALS storage ring, the operation is very stable. Currently real-time response is not required by the controls software. Therefore, most of the programs written in the commissioning phase are still in use, including those written in Toolbook, Excel and Visual Basic. They support a wide variety of fields: the saving and restoring of the machine-device status, the booster energy-ramp linearity-corrections, turning on and off devices in the injector section, timing setting, scraper and TV-paddle controls, and the storage-ring bunch-filling pattern controls. These do not require quick responses and some are complex. For these cases, the tools worked effectively on Windows.

But there are areas where high performance becomes an issue. These areas are covered by programs originally written in compilers. Most of them are object-oriented and use the DMMobj class-library. Many of them, especially those that support magnet operations, had to be updated several times when the operation scheme was changed. As OOP supports such evolution, these modifications were done efficiently. We have seen the benefit of OOP through such experiences.

An operational-context control is being developed. We have a device lock/unlock mechanism that uses shared memory, in the Intelligent Local Controller(ILC) to allocate registers, to count the "*heart beat*" of all magnet power supplies. We did not adopt a static semaphore because of the tolerance issue. Possible conflicts of magnet control, by multiple programs, are being prevented. But, this mechanism is at the device level and is not suitable to control complex operation modes.

A more advanced scheme to organize the machine operation has been investigated[20]. It uses a fully object-oriented approach including an object-oriented design tool and an object-oriented database system. This combination looks very promising.

A model-based control can use both modeling objects and device objects, but it becomes usable only after a model is well calibrated. A model calibration itself is an important task at an early stage of the machine operation. There is a need to provide a model-free control scheme until the calibration is completed[21]. The commissioning of a model is an evolution process and OOP plays an important role.

V. LIMITATIONS OF OBJECT-ORIENTATION

We have confirmed that there are significant benefits from OOP, based on our experiences at ALS. The fields where OOP was applied, have also been functioning very well. To follow are some problems that cannot be ignored.

A. Lack of OO Developers

The first and most crucial problem is the lack of software developers in OOP. It has been said that it takes several months to train staff who already use C. This training period becomes a burden in many cases. Software developer training should be recognized as an important part of software development. Otherwise, it is impossible to set up a group of developers who can coherently work using objects. One example is the pioneering effort at the AGS Booster[22].

B. Reusability and Compatibility

OOP increases software reusability, which means the availability of reusable class libraries. But, a problem occurs as soon as we try to use several libraries. Compatibility among class libraries can be obtained either when they are designed to be compatible or when they are orthogonal to each other. As a result, most class libraries are grouped into families that are exclusive to each other. A very common situation is that the selection of a GUI class determines the family. Unfortunately, this is an inevitable result of the fact that GUI is event-driven, if it contains an application framework.

C. Lack of OO Standards

Class libraries are very useful if they are created according to an existing standard. We can implement standards without knowing the detail by using a reusable class prepared for it. In many cases, the class library becomes an exclusive standard. That is, the lack of reusability and compatibility is mainly due to lack of regulated standards. The reality is that standards are determined and provided by the manufacturer and they have not agreed on a standard, even in GUI classes, especially for UNIX Xwindows.

In addition, there is an intrinsic limit. By its very nature, OOP does not support persistency and concurrency. In other words, OOP itself does not cover databases, multitasking, or networking. They are outside of the language specification and as a result they are not supported by libraries that are not designed for OOP.

Relational databases have been popular, but most of them are proprietary and lack compatibility even with Structured Query Language (SQL). Very recently, SQL added a reasonably accepted interface standard called Open Database Connectivity (ODBC).

There are C++ classes available that wrap ODBC, but the performance of SQL itself will not be sufficient for accelerator control purposes. The standardization has not been sufficient in this field, even before we discuss its object-orientation. Networking has a similar situation. Both are usually treated as parts of various kinds of frameworks that are completely exclusive to each other.

D. Lack of the "Data Module" Concept

A data module is a NODAL[23] concept that lies between the system developers and their users, who are application programmers. It is a kind of software module that is similar to the object, but does not have the inheritance capability. The system developers create data modules as packages of routines. These data modules become new statements of the NODAL interpreter. The users access these modules as a part of the interpreter environment. In the case of Tracy and Gemini, we used a Pascal compiler for this purpose. But, it is not common in the modern GUI environment. Once a GUI application is created, it is not usually expandable. If a function needed by a user is not there, he may have to keep clicking buttons and menus endlessly to reach his goal.

The interface between the system developers and application users can be class libraries. Then the class users must recompile programs for any change they make. The programmability at run time is important and usually not being paid enough attention to by the providers. This is again a consequence of the structured approach, since it requires too much weight on the specifications in advance. OOP improves the situation considerably, but does not support the concerns of the data module concept.

VI. CONCLUSION

When OOP was discussed, the focus was on its benefits. Recently, pitfalls of OOP[24] are being pointed out. We have described the problems with OOP in respect to accelerators. But these are not serious enough for avoiding OOP. In our view the benefits are much greater. Remember that the most important thing in the software construction, for the accelerator project, is to provide programs on time using a reasonable amount of resources. Although we had to make some very practical compromises while using OOP, it improved productivity.

VIII. ACKNOWLEDGMENTS

The author wishes to thank the ALS Controls Section, especially C. Timossi for his continuous support for the device access, the Operators for their feedback on programs, and A. Jackson, R. Keller and K. Van Dongen for their time and advice.

IV. REFERENCES

[1] LBL PUB-5172 Rev. LBL,1986.
A. Jackson, IEEE PAC93, 93CH3279-7(1993)1432
[2] G. Entsminger,"The Tao of Objects", M&T Books, CA, 1990.
[3] B. Meyer, "Object-Oriented Software Construction", Prentice-Hall, NJ, 1988.
[4] B. J. Cox, "Object-Oriented Programming", Addison-Wesley, MA, 1986..
[5] M.Berz, SSC-152, 1988
Leo Michelotti, IEEE PAC89, CH2669-0(1989)839
N. Malitskey, A. Reshetov and Y. Yan,SSCL-659, 1994.
[6] H. Nishimura, N.I.M. A 352, 379, 1994
[7] A. Jackson, E. Forest, H. Nishimura and M. S. Zisman, IEEE PAC89, CH2669-0, 1752, 1989.
[8] H. Nishimura, EPAC 88, (1989)803.
[9] E. Forest and H. Nishimura, IEEE PAC 89, CH2669-0 , 1304, 1989.
[10] R.E.Berry, "Programming Language Translation", Ellis Horwood Ltd., England, 1981.
[11] S. Magyary, IEEE PAC93, 93CH3279-7,1811,1993.
[12] H. Nishimura, Comput. in Phys.6, 456, 1992
[13] H. Nishimura, LSAP-153, LBL Internal Report, 1993
[14] H. Nishimura, LSAP-128, LBL Internal Report, 1992.
[15] C. Kim, LSAP-165, LBL Internal Report, 1994.
[16] G. Portmann and C. Timossi, LSEE-188, LBL Internal Report, 1994
[17] L.R.Dalesio, M.R.Kraimer and A.J.Kozubal, ICALEPCS 91, KEK Proceedings 92-15, 278, KEK, Japan.
[18] C. Timossi, LSEE 121, LBL Internal Report,1995.
[19] J. A. Hinkson, M. Chin, C. H. Kim, H. Nishimura, to be published in the proceedings of EPAC94.
O. Jones, private communication
[20] H. Nishimura, C. Timossi, M. Valdez, these proceedings.
[21] H. Nishimura, L.Schachinger and H. Ogaki, these proceedings.
[22] J. F. Skelly, ICALEPCS 91, KEK Proceedings 92-15, 500, KEK, Japan.
[23] M.C. Crowley-Milling and G.C. Shering, CERN 78-07, 1978.
[24] B. E. Webster, "Pitfalls of Object-Oriented Development", M&T Books, NY, 1995.